International Journal of Computer Engineering and Technology (IJCET)
Volume 14, Issue 3, Sep-Dec 2023, pp. 140-148, Article ID: IJCET_14_03_014
Available online at https://iaeme.com/Home/issue/IJCET?Volume=14&Issue=3
ISSN Print: 0976-6367 and ISSN Online: 0976–6375

OPTIMIZING AWS LAMBDA COLD STARTS THROUGH PRIMING: A TECHNICAL EXPLORATION

Balasubrahmanya Balakrishna
USA

ABSTRACT

This technical paper explores how Priming can be a deliberate tool to reduce cold start[1] time in AWS Lambda serverless services. Cold starts, indicating initial delays when executing a function after inactivity, often pose challenges in serverless systems. Using a heavy-weight dependency injection design or framework in Java systems exacerbates this difficulty. The article analyzes situations where desired response times are not attained, uncovering optimization insights even when devising concurrency models such as provisioned concurrency and Snapstart, which aim to reduce cold starts, are adopted. This article investigates the adoption of priming methods to decrease cold start times, thereby improving the overall performance of serverless apps.

The author, coming from an AWS and Java background, is committed to using these technologies to express the concept throughout.

Keywords: AWS Serverless Lambda, Provisioned Concurrency, SnapStart, Cloudwatch, Spring Boot, JVM, AWS Fargate, AWS EC2, AWS ECS, HTTP Client, AWS DynamoDB

Cite this Article: Balasubrahmanya Balakrishna, Optimizing Aws Lambda Cold Starts Through Priming: A Technical Exploration, International Journal of Computer Engineering and Technology (IJCET), 14(3), 2023, 140-148.
https://iaeme.com/Home/issue/IJCET?Volume=14&Issue=3

INTRODUCTION

Understanding and minimizing cold start delays is becoming increasingly important as enterprises rely on serverless computing for mission-critical applications. This article provides developers, architects, and system administrators with the information and tools they need to deploy priming approaches intelligently, ultimately increasing AWS Lambda performance and enhancing the user experience for serverless apps. Readers will get significant insights into the subtleties of Priming and its disruptive potential in serverless computing due to this technical exploration.

"Priming" refers to putting data or initializing resources into memory before their initial use. This proactive strategy reduces execution delays by ensuring that the assets required when a function is activated are readily available. While this strategy applies to many applications, its benefits are especially noticeable in the serverless Lambda space, where AWS periodically reinitializes Lambda.

Consider cases where an AWS Lambda function uses external resources, such as database connections or HTTP clients. In database connection, Priming could entail creating and caching the connection during an "INIT" phase[1] for a Lambda function that interacts with a database. When the Lambda function is later invoked ("Warm Start")[1], the database connection is already in memory, minimizing the time required to create the connection on the fly. Priming is especially useful in cases where low-latency database interactions are critical. Similarly, Consider a Lambda function that makes HTTP queries to external APIs or services. Priming would entail initializing and storing the HTTP client connection and any necessary authentication tokens or variables. When a Lambda function is invoked warmly, the HTTP client connection is already in memory, saving time that might otherwise be spent setting up the connection dynamically. Priming is helpful for applications that require immediate and responsive contact with external services.

A. Lambda with Provisioned Concurrency

Consider the following diagram (Fig.1), which depicts a Lambda running a Java application with Provisioned Concurrency. This diagram illustrates the basic activities of a

Lambda function utilizing Provisioned Concurrency[3]. For simplicity, we've limited the concurrency to one container by deploying the function with a Provisioned Concurrency value of 1. With this configuration, the Lambda function avoids cold starts, which occur when initial requests wait for the container or execution environment[4] to kick off.

Certain classes not being fully initialized may still prolong the initial response times for the first request despite the lack of cold starts. This behavior is similar to what you could see with AWS Serverful Compute and other AWS Serverless container-based services like Fargate. AWS initiates periodic reinitialization of Lambda containers, which occurs every hour based on observations. This recurrent reinitialization can result in heightened response times for the first request.

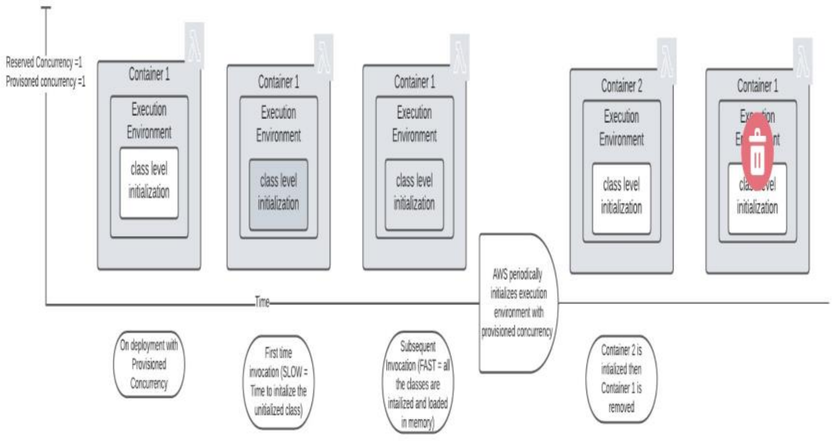
Reserved Concurrency =1
Provisioned concurrency =1
Container 1
Execution Environment
class level initialization
Container 1
Execution Environment
class level initialization
Container 1
Execution Environment
class level initialization
Container 2
Execution Environment
class level initialization
Container 1
Execution Environment
class level initialization
Time
AWS periodically initializes execution environment with provisioned concurrency
On deployment with Provisioned Concurrency
First time invocation (SLOW = Time to initalize the unitialized class)
Subsequent Invocation (FAST = all the classes are intailized and loaded in memory)
Container 2 is intialized then Container 1 is removed

Fig. 1. AWS Lambda with provisioned concurrency and periodic reinitialization

B. Lambda with SnapStart

The same logic applies to a Lambda function that uses SnapStart[2]. The core mechanisms of a SnapStart-enabled Lambda function are depicted in the diagram (Fig. 2) below. When a SnapStart-enabled Lambda function faces a cold start, it uses a previously saved snapshot of the function to reduce the cold start time drastically. However, if the snapshotted Lambda function does not have the necessary data in its memory, an additional penalty may be imposed, resulting in increased latency for the original request.

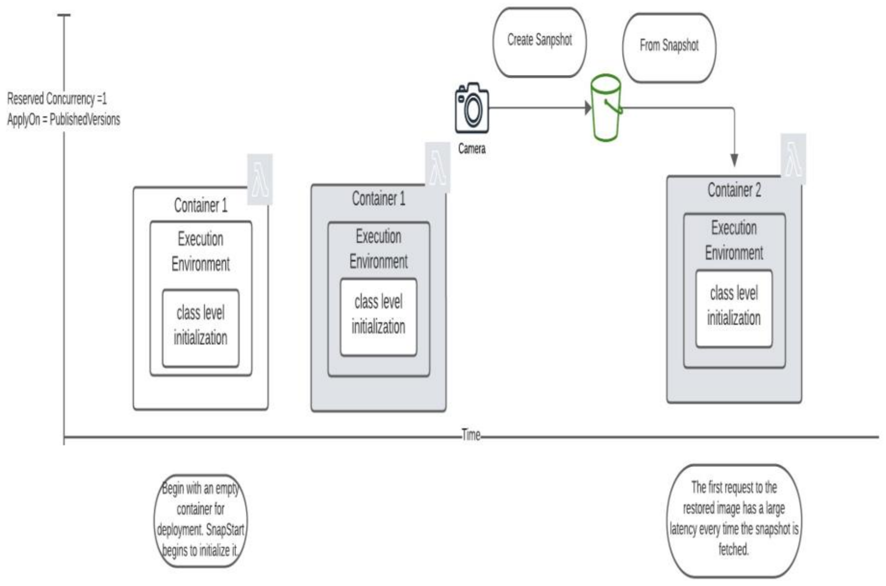
Reserved Concurrency =1
ApplyOn = PublishedVersions
Create Sanpshot
From Snapshot
Camera
Container 1
Execution Environment
class level initialization
Container 1
Execution Environment
class level initialization
Container 2
Execution Environment
class level initialization
Time
Begin with an empty container for deployment. SnapStart begins to initialize it.
The first request to the restored image has a large latency every time the snapshot is fetched.

Fig. 2. AWS Lambda with SnapStart and periodic reinitialization

PRIMING APPROACHES

We may address the above issue: one requires specifically priming a particular AWS resource. At the same time, the other involves a broader technique of priming any resource. Let us systematically examine both options outlined below:

A. Priming an AWS Resource

Consider a scenario in which SnapStart-enabled Lambda functions display prolonged cold start times primarily due to DynamoDB configuration. We can strategically initialize the DynamoDB client during the SnapStart snapshot creation process to remedy this. Priming is accomplished by including a "static" block within the handler class. The following section (Fig. 3) presents a systematic strategy for implementing client priming in this specific context:

```
// imports…

public class DynamoDbLambdaHandler implements
RequestHandler<Object, Object> {

    private static final DynamoDbClient client;
    private static final String tableName =
System.getenv("TABLE_NAME");
static {
        client =
DynamoDbClient.builder().build();
initializePriming();
    }

private static void initializePriming() {
        boolean isPrimingEnabled =
Boolean.parseBoolean(System.getenv("ENABLE_PRIM
ING"));
        if (isPrimingEnabled) {
            primeDynamoDb();
        }
    }

private static void primeDynamoDb() {
        try {
            Map<String, AttributeValue> key =
Collections.singletonMap("PK",
AttributeValue.builder().s("foobar").build());
            GetItemRequest request =
GetItemRequest.builder().key(key).tableName(tab
leName).build();
            Map<String, AttributeValue> item =
client.getItem(request).item();
            System.out.println("Priming
DynamoDB with item: " + item);
        } catch (DynamoDbException |
RuntimeException e) {
            System.err.println("Error priming
DynamoDB: " + e.getMessage());
            e.printStackTrace();
        }
    }

 @Override
 public Object handleRequest(Object input,
Context context) {
        Instant startTime = Instant.now(); //

  try {
DynamoDbLambdaHandler.primeDynamoDb();

System.out.println("Executing long-running
operation...");
            Thread.sleep(30000);
            return "Lambda function executed
successfully!";
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            System.err.println("Thread sleep
interrupted: " + e.getMessage());
            e.printStackTrace();
            return "Lambda function
interrupted!";
        } finally {
            Instant endTime = Instant.now();
Duration executionTime =
Duration.between(startTime, endTime);
            System.out.println("Lambda
execution time: " + executionTime.toMillis() +
" ms");
        }
    }
}
```

Fig. 3. Code snippet: Priming AWS Resource

1. Static Initialization Block: Based on the ENABLE_PRIMING environment setting, this block initializes the DynamoDbClient and executes client priming.
2. initializePriming Method: This method determines whether client priming is enabled and calls the primeDynamoDb method accordingly.
3. primeDynamoDb Method: This function initializes DynamoDB by issuing a sample GetItem request with the provided key ("foobar").
4. handleRequest Method: The RequestHandler interface requires this handler method for the Lambda function.
5. Capture Start Time: Records the beginning time of the Lambda function execution.
6. Invoke Client Priming: The primeDynamoDb method performs client priming.
7. Simulate Long-Running Operation: A Thread.sleep simulates a long-running operation (30 seconds).
8. Main Logic: A comment placeholder represents the basic logic of the Lambda function using the DynamoDB client.
9. Capture End Time: Records the Lambda function execution time and the total execution time.

We force the DynamoDB setup before snapshot creation by including a dummy DynamoDB request within the static block. As a result, the snapshots will consist of this pre-executed initialization by default. With this configuration, a series of tests were run by switching the ENABLE_PRIMING flag between false (depicted in Fig. 4) and true (depicted in Fig. 5) for 1000 repetitions, obtaining the following owing results: The test results show a considerable improvement in response time, with the median lowering from *1903 ms* to *195 ms*, equating to an improvement of ≅ *90%*.

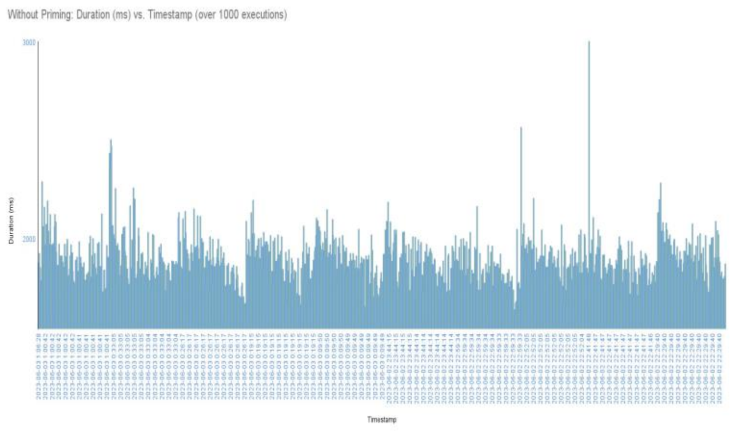

Fig.4 Test result: Priming disabled on AWS Resource

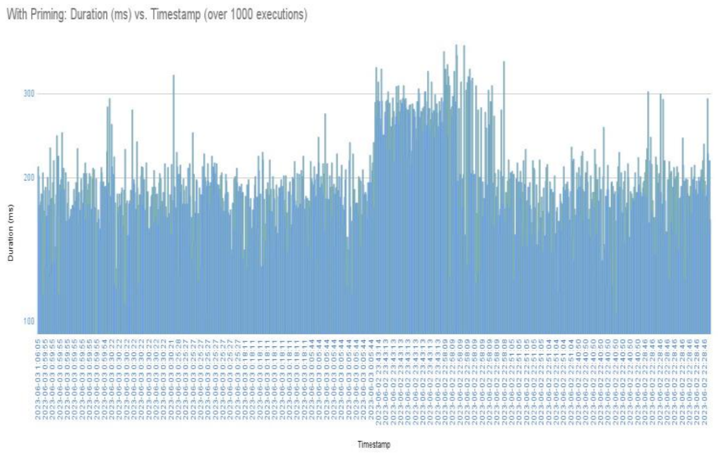

Fig.5 Test result: Priming enabled on AWS Resource

The test results show a considerable improvement in response time, with the median lowering from *1903 ms* to *195 ms*, equating to an improvement of ≅ *90%*.

B. Priming any Resource

Even when the use case does not need a connection with AWS resources such as DynamoDB, an increased delay can be observed for the initial request. When deploying a Spring Boot Lambda function with Provisioned Concurrency, response times may increase noticeably due to the reinitialization of Lambda Provisioned Concurrency instances.

In such cases, sending a dummy request to the Lambda function during the "INIT" phase helps reduce latency. The following code snippet (depicted in Fig. 6) presents a systematic strategy for implementing client priming in this context.

```
//imports...

public class StreamLambdaHandler implements
RequestStreamHandler {
```

```

    private static final
SpringBootLambdaContainerHandler<AwsProxyReques
t, AwsProxyResponse> handler;
    private static final String
MOCK_LAMBDA_REQUEST_STRING =
generateMockLambdaRequestString();

    static {
        try {
            handler =
SpringBootLambdaContainerHandler.getAwsProxyHan
dler(PrimingWebApplication.class);
        } catch
(ContainerInitialisationException e) {
            throw new RuntimeException("Could
not initialise Spring Boot application", e);
        }
    }

    public StreamLambdaHandler() {
        try {
            initializeAndHandleRequest();
        } catch (Exception e) {

LoggerFactory.getLogger(StreamLambdaHandler.cla
ss).error("Error in constructor", e);
        }
    }

    /**
     * Initializes and handles the request,
performing client priming if needed.
     *
     * @throws IOException If an I/O error
occurs.
     */
    private void initializeAndHandleRequest()
throws IOException {
        try (InputStream targetStream = new
ByteArrayInputStream(MOCK_LAMBDA_REQUEST_STRING
.getBytes("UTF-8"));
             OutputStream output = new
ByteArrayOutputStream()) {

            System.out.println("In
StreamLambdaHandler Constructor");
            this.handleRequest(targetStream,
output, new MyAwsContext());
            System.out.println(output);
        }
    }

    /**
     * Handles an AWS Lambda request,
performing client priming and capturing
duration.
     *
     * @param inputStream  The input stream
containing the Lambda request.
     * @param outputStream The output stream to
write the Lambda response.
     * @param context      The execution
context of the Lambda function.
     * @throws IOException If an I/O error
occurs.
     */
    @Override
    public void handleRequest(InputStream
inputStream, OutputStream outputStream, Context
context) throws IOException {
        Instant start = Instant.now();
```

```
        handler.proxyStream(inputStream,
outputStream, context);
        Instant end = Instant.now();
        long durationMillis =
Duration.between(start, end).toMillis();

        // Print the duration based on the
context.
        if (context != null &&
context.getInvokedFunctionArn().equals("your-cu
stom-string")) {
            System.out.printf("Client Init
Duration: %d ms%n", durationMillis);
        } else {
            System.out.printf("Client Request
Duration: %d ms%n", durationMillis);
        }

        try {
            // Pause for 1 minute (60 seconds)
to force the use of other lambda instances.
            Thread.sleep(60000);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }

    /**
     * Generates a mock Lambda request string
for testing purposes.
     *
     * @return The generated mock Lambda
request string.
     */
    private static String
generateMockLambdaRequestString() {
        // Modify this method to generate the
request string based on your requirements.
        return "{\n" +
               // ... (your request
structure)"}";
    }

    // Other methods…
}
```

Fig.6 Code snippet: Priming Resource broadly

1. Constructor: Initializes and Handles the request, emphasizing potential client priming.
2. initializeAndHandleRequest: This method initializes and handles the request method, emphasizing client priming.
3. handleRequest: Handles an AWS Lambda request, including client priming and duration capture.
4. generateMockLambdaRequestString: Creates a simulated Lambda request string for testing.

The outcomes of this test (depicted in Fig. 7) demonstrate a significant enhancement in response time, decreasing from 6000 ms to 60 ms, which translates to a tenfold improvement.

RATIONALE

In contemplating this approach, one might question the apparent unconventional nature of making dummy requests to AWS resources or the application itself. Acknowledged as a somewhat uncommon practice, initializing specific resources sometimes necessitates sending a dummy request. While this method lacks elegance, its effectiveness makes it a valid consideration, introducing a trade-off that demands evaluation when assessing the solution.

CONSIDERATIONS

Caution is paramount when employing Priming due to potential side effects. When interacting with AWS resources, unintended modifications to the downstream state may occur, posing challenges for the application. For instance, initializing DynamoDB by inserting a fake item may incur additional costs and introduce data into the table that could disrupt normal app execution. It is crucial to ensure that modified data does not cause issues or turn off the Priming portion if opting for the general Priming solution (where the app calls itself).

CONCLUSION

Despite these considerations, Priming is a powerful method for reducing first-request latency. Priming, combined with SnapStart or Provisioned Concurrency, advocates addressing the core pain point of Cold Starts from a code-centric perspective rather than the execution environment. Evaluating first-request response times becomes critical; Priming may be helpful if they are higher than typical and dealing with high TPS and low latent applications. Also, priming can lower the cost as it reduces the execution time of a function.

REFERENCES

[1] AWS (n.d.). Lambda execution environments. AWS Lambda Documentation. https://docs.aws.amazon.com/lambda/latest/operatorguide/execution-enviro nments.html

[2] AWS (n.d.). Reducing java cold starts on aws lambda functions with snapstart. AWS Lambda Documentation.

https://aws.amazon.com/blogs/compute/reducing-java-cold-starts-on-aws-la mbda-functions-with-snapstart/

[3] AWS (n.d.). Configuring Provisioned Concurrency. AWS Lambda Documentation.

https://docs.aws.amazon.com/lambda/latest/dg/provisioned-concurrency.ht ml

[4] AWS (n.d.). Lambda execution environment. AWS Lambda Documentation.

https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtime-environmen t.html

[5] AWS (n.d.). Concurrency. AWS Lambda Documentation.

https://docs.aws.amazon.com/lambda/latest/dg/API_Concurrency.html

[6] AWS (n.d.). Operating Lambda: Performance optimization – Part 1.
AWS Lambda Documentation.
https://aws.amazon.com/blogs/compute/operating-lambda-performance-opti mization-part-1/

[7] AWS (n.d.). Operating Lambda: Performance optimization – Part 2.
AWS Lambda Documentation.
https://aws.amazon.com/blogs/compute/operating-lambda-performance-opti mization-part-2/

AUTHOR INFORMATION

Balasubrahmanya Balakrishna is a Senior Lead Software Engineer, serverless-focused strategy leader, open-source enthusiast, and cloud-native advocate.

Citation: Balasubrahmanya Balakrishna, Optimizing Aws Lambda Cold Starts Through Priming: A Technical Exploration, International Journal of Computer Engineering and Technology (IJCET), 14(3), 2023, 140-148.

Article Link:
https://iaeme.com/MasterAdmin/Journal_uploads/IJCET/VOLUME_14_ISSUE_3/IJCET_14_03_014.pdf

Abstract Link:
https://iaeme.com/Home/article_id/IJCET_14_03_014